Semantic Order Compatibilities and Their Discovery

by

Melicaalsadat Mirsafian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Ordered domains such as numbers and dates are common in real-life datasets. The SQL standard includes an ORDER BY clause to sort the results, and there has been research work on formalizing, reasoning about, and automatically discovering order dependencies among columns in a table. However, a crucial assumption made in research and practice is that the order over a column is syntactic: numbers are ordered numerically, strings lexicographically and dates chronologically. To the best of our knowledge, this work is the first to relax this assumption. We present a generalized definition of order compatibilities that allows *semantic* orders such as (low, medium, high) or (excellent, very good, good, average, poor). We show that in general, validating whether there exists a semantic order relationship between columns is NP-complete, with some tractable special cases. We give an algorithm to automatically discover semantic order relationships in the data, we provide examples of interesting orders found by our algorithm that were missed by existing algorithms, and we show that the NP-complete validation cases do not occur frequently in practice.

Acknowledgements

I would like to thank all the people who made this thesis possible. Especially my supervisor, Lukasz Golab, for his support and guidance throughout this process. I would also like to thank Jarek Szlichta, Parke Godfrey, Mehdi Kargar, and Divesh Srivastava for offering me their time, support, and ideas and helping form this thesis and establish *Semantic Order Compatibilities* as a novel idea.

Dedication

I dedicate this work to all beings in the universe who have made me into who I have become. Especially my family for giving me life and being an endless source of unconditional love throughout every stage of my life. I thank my soulmate and my sister Leila and my mother Akram for being the amazing empowering feminists that they are. I thank my father Hassan for always having me around when he was fixing something around the house and asking me to help him when I was just a child. He gave me my love for crafting and engineering.

I want to thank all the friends I found during my time as a graduate student, for all the wonderful and deep moments we collected together. I have been blessed with many great friendships and naming everyone would be out of the capacity of this document. However, Tosca, Amine and Josh have been my safe haven and chosen family these past few months and I am infinitely grateful for their company. I thank them for loving me and I thank them for letting me love them.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

2 Related Work 4

3 Preliminaries and Definitions 6

4 Semantic Order Compatibilities 10

4.1 A General Definition for Order Compatibilities 10

4.2 Discovering Semantic Order Compatibilities: Lattice Traversal 14

4.3 Validating Syntactic Order Compatiblities 15

4.4 Validating Semantic Order Compatibilities: Special Case 16

4.4.1 Validating Each Equivalence Class 17

4.4.2 Validating $SemOC_{sc}$ Over All Equivalence Classes 19

4.5 Validating Semantic Order Compatibilities: General Case 19

4.5.1 Validating a single equivalence class 20

4.5.2 Chain Derivation 23

4.5.3 Merge and Validate Polarity Groups 25

4.5.4 Merge Polarity Groups With Loose Connections 26

4.6 Complexity Analysis 27
4.6.1 Validation of a Single $SemOC_{sc}$ 27
4.6.2 Validation of a Single $SemOC_{gc}$ 28

5 Experiments **31**
5.1 Data Characteristics 33
5.2 Qualitative Experiments 33
5.3 Quantitative Experiments 34
5.3.1 $SemOC_{gc}$ Validation 35
5.3.2 NP-complete Cases in Practice 37
5.3.3 Validation Times for Different Types of Dependencies 37

6 Conclusions and Future Work **40**

References **41**

List of Tables

1.1 Grades in different scales 2

1.2 Overlapping months from the Persian and Gregorian calendars 3

4.1 A table containing a 3-fan-out, corresponding to Figure 4.3 11

4.2 Grades in different scales in Canada and in Iran 15

4.3 numeric representation of Gregorian months and their overlap with Persian months 18

4.4 Table 4.3 transformed 18

4.5 Table 5.4 transformed 19

4.6 Gregorian and Persian months corresponding to a set of dates 20

5.1 $SemOC_{gc}$ candidates from the *flights* dataset exceeding 400 iterations in their validation 31

5.2 skipped candidates from rows 1 and 2 of Table 5.1, as considered on the next lattice level 32

5.4 a sample set of ages and their corresponding age range 34

5.3 Overlapping_months from the Persian and Chinese calendar 39

List of Figures

4.1 Set containment lattice for the attributes in Table 4.2 (. 16

4.2 biGraphs representing co-occurence of values of *gmonth* and *pmonth* from Table 4.6 for each equivalence class in $\{gyear\}$ 21

4.3 biGraph corresponding to Table 4.1 21

4.4 Set containment lattice for 29

5.1 Scatter plots showing the time taken for each $SemOC_{gc}$ candidate validation. 36

5.2 Boxplots showing the time taken (logarithmic) to validate different types of order candidates on different datasets. 38

Chapter 1

Introduction

Datasets with ordered domains are common in many applications: for example, numeric domains are common in machine-generated data streams, while numeric quantities, dates and timestamps are common in data warehouses. In recognition of the importance of ordered domains, the SQL standard includes an ORDER BY clause to sort the output, and there exists research on formalizing order relationships in the data through *Order Dependencies* [3, 6, 9]. Intuitively, an order dependency asserts that sorting a table according to some column(s) implies that the table is also sorted according to another column.

For a simple example, consider a financial table with two columns (among many others): salary and tax. If tax is a fixed percentage of salary, then sorting this table by salary also sorts it by tax, and vice versa. For another example, consider an e-commerce database with order timestamps and order numbers. If order numbers are assigned sequentially over time, then there is an order dependency between order timestamps and order numbers (and, again, vice versa).

In practice, order dependencies are not always known or included with a database schema, motivating the need to automatically discover them from the data. There has been some recent work on order dependency discovery [5, 9].

However, in the context of order relationships, an important assumption made in theory and practice is that there is a natural order for a given domain: numeric order for numbers, lexicographic order for strings, chronological order for dates and timestamps, etc. To the best of our knowledge, this work is the first to relax this assumption and ask the following question: *can we discover new or latent orders in the data*? For example, ordinal attributes and strings representing numbers or ranges of numbers can create new interesting orders, as motivated below.

Table 1.1: Grades in different scales

#	country	grade_num	grade_letter	grade_desc
1	Canada	12	F	Fail
2	Canada	20	F	Fail
3	Canada	70	C	Good
4	Canada	75	B	Very Good
5	Canada	85	B	Very Good
6	Canada	95	A	Excellent

Consider Table 1.1 showing academic grades in different formats: numeric grade, letter grade, and a textual grade description. If we sort the table by grade_num ascendingly, we obtain a monotonically non-increasing order over the values of grade_letter. This relationship relies on lexicographic order and can be detected by existing order dependency discovery algorithms. However, if we sort the table by grade_num, we find a sort over the values of grade_desc, the grade descriptor, that is not lexicographic, yet semantically meaningful: $(Fail < Good < VeryGood < Excellent)$.

Now consider Table 1.2, with g_month representing months in the Gregorian calendar, and p_month and g_date representing the corresponding month in the Persian calendar, and its starting date. Based on the co-occurrence of values of p_month and g_month, if we use the non-lexicographic (but meaningful) order over g_month of $(January < February < March < ...)$ and sort the table by this new order, we will find a non-lexicographic (but meaningful) order over p_month of $(Azar < Aban < Mehr < ...)$. Existing algorithm based on lexicographic orders would miss these interesting relationships.

Prior work on order relationships in data has focused on numeric or lexicographic orders. Our goal in this work is to automatically find meaningful *semantic* orders such as those discussed above. We make the following contributions:

1. We formalize a general definition of *semantic order compatibility* that allows meaningful non-lexicographic orders.

2. We show that verifying whether a semantic order exists between a pair of columns is NP-complete in the general case but polynomially solvable in some special cases.

3. We present an algorithm to discover semantic orders from data by building on the state-of-the-art algorithm for lexicographic order discovery from [9]. Using real datasets, we show that the NP-complete cases rarely occur in practice.

Table 1.2: Overlapping months from the Persian and Gregorian calendars

#	g_month	p_month	g_date
1	December	Azar	2018-12-08
2	November	Azar	2018-11-21
3	November	Aban	2018-11-08
4	October	Aban	2018-10-31
5	October	Mehr	2018-10-10
6	September	Mehr	2018-09-30
7	September	Shahrivar	2018-09-10
8	August	Sharivar	2018-08-31
9	August	Mordad	2018-08-12
10	July	Mordad	2018-07-31
11	July	Tir	2018-07-22
12	June	Tir	2018-06-30
13	June	Khordad	2018-06-21
14	May	Khordad	2018-05-31
15	May	Ordibehesht	2018-05-21
16	April	Ordibehesht	2018-04-30
17	April	Farvardin	2018-04-17
18	March	Farvardin	2018-03-31
19	March	Esfand	2018-03-18
20	February	Esfand	2018-02-28
21	February	Bahman	2018-02-17
22	January	Bahman	2018-01-31
23	January	Dey	2018-01-20

4. Using our algorithm, we show examples of interesting semantic orders found in real datasets.

The remainder of this paper is structured as follows. Chapter 2 includes a summary of prior work on order dependencies. Chapter 3 provides preliminaries and background information. In Chapter 4, we formalize semantic order compatibilities and give an algorithm for their discovery. We show experimental results in Chapter 5 and we conclude in Chapter 6 with directions for future work.

Chapter 2

Related Work

In SQL, order is expressed using a list of attributes, an *order specification*, in the order-by clause. This list of attributes denotes a nested sort over the tuples, i.e. sorting the tuples by the first attribute in the list, breaking ties by the second attribute in the list and so on. The SQL order-by clause orders tuples based on the ascending or descending lexicographic (for strings), numeric (for numbers), or chronological (for dates) order.

The notion of order was formally introduced in the context of databases by Ginsburg and Hull [3] in 1983. They examined and formally defined dependencies which incorporate information involving order, calling them *order dependencies* (ODs).

In 2001, Ng [6] extended the relational data model to incorporate partially ordered domains, calling it the ordered relational model. They studied lexicographic order dependencies, as a generalized form of functional dependencies that can capture monotonically non-decreasing relations between two sets of values projected onto some attributes in a relation. An OD such as $\mathbf{X}$ *orders* $\mathbf{Y}$ intuitively means that if tuples in a relation are sorted by $\mathbf{X}$, then they will also be sorted by $\mathbf{Y}$.

In 2012, Szlichta et al. [8] studied lexicographic orderings in further detail and provided a sound and complete axiomatization for ODs. Szlichta et al. [10] provided a set of inference rules for ODs and proved that the problem of inferring ODs is co-NP-complete.

Some dependencies, mainly key constraints, that hold in a database are known at design time, but that is not always the case, prompting the development of dependency discovery techniques [1] such as TANE [4] for FD discovery. TANE searches through a set-containment lattice of the attributes in a schema, discovering FDs that hold between different attributes. ORDER [5] discovers ODs by traversing a lattice search space of lists of attributes in a schema.

Szlichta et al. [9] showed that attribute lists are not necessary for ODs and provided a system of sound and complete set-based axioms. Their axiomatization allowed for a much smaller search space to be explored during OD discovery, cutting the time complexity down to exponential in the number of attributes compared to factorial complexity in the number of attributes for ORDER [5].

All prior work on order dependencies has focused on lexicographic ordering. In this work we provide a novel definition for order compatibility that includes semantic orders.

Chapter 3

Preliminaries and Definitions

We use the following notation:
$\mathbf{R}$ denotes a *relation* and $\mathbf{r}$ denotes a particular instance of relation $\mathbf{R}$.

Each of A, B, and C denotes an *attribute* in the relation while t and s refer to *tuples*. t_{A} is the *projection* of tuple t on attribute A.

$\mathcal{X}$ and $\mathcal{Y}$ refer to *sets* of attributes and $t_{\mathcal{X}}$ denotes the projection of tuple t on set $\mathcal{X}$. The empty set is denoted as $\{\}$.

Lists of attributes are denoted as $\mathbf{X}$, $\mathbf{Y}$ or $\mathbf{Z}$. They each may represent the empty list denoted as $[\,]$. An explicit list is denoted as $\mathsf{A}, \mathsf{B}, \mathsf{C}$. $[\mathsf{A} \mid \mathbf{T}]$ refers to a list whose first attribute is A (*head*) and the rest of its attributes are denoted as list $\mathbf{T}$(*tail*).

Definition 3.0.1 *We refer to the default ordering over the data as* syntactic order. *This is the order considered by SQL in the order-by clause and is ascending in the lexicographic ordering for strings, numeric for numbers, and chronological for dates.*

Definition 3.0.2 *An attribute set $\mathcal{X}$ partitions tuples into* equivalence classes *[4]. The equivalence class of a tuple $t \in \mathbf{r}$ with respect to an attribute set $\mathcal{X} \in \mathbf{R}$ is denoted by $\mathcal{E}(t_{\mathcal{X}})$ $= \{s \in \mathbf{r} \mid s_{\mathcal{X}} = t_{\mathcal{X}}\}$. A* partition *of $\mathbf{r}$ over $\mathcal{X}$ is the set of equivalence classes, $\Pi_{\mathcal{X}} = \{\mathcal{E}(t_{\mathcal{X}})$ $\mid t \in \mathbf{r}\}$.*

For instance in Table 1.1, $\mathcal{E}(t4_{\{\mathsf{grade\_desc}\}}) = \mathcal{E}(t5_{\{\mathsf{grade\_desc}\}})$, because $t4_{\{\mathsf{grade\_desc}\}} = t5_{\{\mathsf{grade\_desc}\}}$ $=$ "F" and $\Pi_{\{\mathsf{grade\_desc}\}} = \{\{t4, t5\}, \{t3\}, \{t1, t2\}, \{t6\}\}$.

Definition 3.0.3 *A* sorted partition τ_{A} *is a list of equivalence classes according to the ordering imposed on the tuples by A. For instance, in Table 5.3, $\tau_{\mathsf{grade\_letter}} = [\{t1, t2\}, \{t3\}, \{t4, t5\}, \{t6\}]$.*

Definition 3.0.4 *On a set $\mathcal{P}$, a binary relation $\preceq$ is a* partial order *if for all items $a, b, c \in \mathcal{P}$ the following statements hold:*

- ***Antisymmetry**: If $a \preceq b$ and $b \preceq a$ then $a = b$*
- ***Transitivity**: If $a \preceq b$ and $b \preceq c$ then $a \preceq c$*
- ***Reflexivity**: $a \preceq a$*

A partial order is a total order *(also called a* chain*) if the **connexity** property holds as well, meaning that all items in $\mathcal{P}$ are comparable. A weak total order is an ordering for which the **transitivity**, **reflexivity**, and **connexity** properties hold, but not the **antisymmetry** property.*

An *order specification* is a list of attributes defining a lexicographic order, as in the SQL order-by clause. An order specification such as `order by A, B` requires the tuples to be sorted by A first, then within the set of tuples with a particular value of A, sorted by B. This is a *nested* sort and it is based on *lexicographic* ordering.

We use the operator '$\preceq_{\mathbf{X}}$' to define a *weak total order* over any set of tuples, where $\mathbf{X}$ denotes an order specification.

Definition 3.0.5 *Let $\mathbf{X}$ be a list of attributes. For two tuples r and s, $\mathcal{X} \subseteq \mathbf{R}$, $r \preceq_{\mathbf{X}} s$ if*

- $\mathbf{X} = [\,]$*;* or
- $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}]$ *and $r_{\mathsf{A}} < s_{\mathsf{A}}$; or*
- $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}]$*, $r_{\mathsf{A}} = s_{\mathsf{A}}$, and $r \preceq_{\mathbf{T}} s$.*

Let $r \prec_{\mathbf{X}} s$ if $r \preceq_{\mathbf{X}} s$ but $s \not\preceq_{\mathbf{X}} r$.

The definition of an order dependency (OD) as provided by Szlichta et al. [8] is as follows:

Definition 3.0.6 *([8], Definition 4) Let $\mathbf{X}$ and $\mathbf{Y}$ be order specifications. $\mathbf{X} \mapsto \mathbf{Y}$ (read: $\mathbf{X}$* orders *$\mathbf{Y}$) denotes an* order dependency *if for all $r, s \in \mathbf{r}$, $r \preceq_{\mathbf{X}} s$ implies $r \preceq_{\mathbf{Y}} s$.*

The OD $\mathbf{X} \mapsto \mathbf{Y}$ means that $\mathbf{Y}$'s values are monotonically non-decreasing with respect to $\mathbf{X}$'s values. In other words, if we order the tuples based on $\mathbf{X}$, they will also be ordered with respect to $\mathbf{Y}$.

Definition 3.0.7 *([8], Definition 5) Two order specifications* $\mathbf{X}$ *and* $\mathbf{Y}$ *are* order compatible*, denoted as* $\mathbf{X} \sim \mathbf{Y}$*,* iff $\mathbf{XY} \mapsto \mathbf{YX}$ *and* $\mathbf{YX} \mapsto \mathbf{XY}$*.*

Intuitively, if $\mathbf{X}$ and $\mathbf{Y}$ are order compatible over a given relation, it means that sorting the tuples by $\mathbf{X}$ and breaking ties by $\mathbf{Y}$ is equivalent to sorting by $\mathbf{Y}$ and breaking ties by $\mathbf{X}$.

Example 3.0.1 *In the TPC-DS benchmark*[1]*,* [d_month] $\sim$ [d_week] *is a valid order compatiblity, where* d_month *is the month number within a year (from 1 to 12) and* d_week*is the week number within a year (from 1 to 53). This means that sorting the tuples by* d_month *and breaking ties by* d_week *is equivalent to sorting by* d_week *and breaking ties by* d_month*.*

Definition 3.0.8 *We write* $\mathcal{X} \rightarrow \mathcal{Y}$ *and say that* $\mathcal{X}$ *functionally determines* $\mathcal{Y}$ *if no two tuples in the table that agree on their values of* $\mathcal{X}$ *have different* $\mathcal{Y}$ *values.*

Theorem 1 *([8], Theorems 13 and 15) For every instance* $\mathbf{r}$ *of relation* $\mathbf{R}$*,* $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\mathcal{X} \rightarrow \mathcal{Y}$ *and* $\mathbf{X} \sim \mathbf{Y}$*.*

Definition 3.0.9 *Szlichta et al. defined a set-based canonical form for ODs, which consists of two types of dependencies ([9], Definition 6):*

1. $\mathcal{X}$*:* $[\,] \mapsto_{cst}$ A *(read:* A *is a* constant *in the context of* $\mathcal{X}$*). An attribute* A *is a* constant *within each equivalence class w.r.t.* $\mathcal{X}$ *if* $\mathbf{X'} \mapsto \mathbf{X'}$A *for any permutation* $\mathbf{X'}$ *of* $\mathbf{X}$*. This is equivalent to the FD* $\mathcal{X} \rightarrow$ A*.*
2. $\mathcal{X}$*:* A $\sim$ B *(read:* A *is* order compatible *with* B *in the context of* $\mathcal{X}$*). Two attributes* A *and* B *are order-compatible within each equivalence class with respect to* $\mathcal{X}$*, denoted as* $\mathcal{X}$*:* A $\sim$ B*, if* $\mathbf{X'}$A $\sim$ $\mathbf{X'}$B *for any permutation* $\mathbf{X'}$ *of* $\mathbf{X}$*.*

[1] www.tpc.org/tpcds/

The set $\mathcal{X}$ in the above definitions is called the *context*. The context represents the factored-out attributes on which the left-hand-side and the right-hand-side of the dependency are equivalent. While the context may consist of a set of attributes, the left and right hand side of a dependency in this canonical form consists of one attribute each.

Of the two types of canonical ODs defined above, one expresses FDs without any order specification, while the other one expresses order compatibilities. In the remainder of this paper, we focus on order compatibilities (OCs) and their generalization to semantic orders.

Definition 3.0.10 *We say that* $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$ *is a minimal OC, if for every* $\mathsf{C} \in \mathcal{X}$ *the OC* $\mathcal{X} \backslash \mathsf{C} : \mathsf{A} \sim \mathsf{B}$ *is not valid.*

Chapter 4

Semantic Order Compatibilities

4.1 A General Definition for Order Compatibilities

Definition 4.1.1 *In a general sense of order compatibility, we say that* $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$ *if there exist one-on-one and onto functions,* $lm : \mathsf{A} \rightarrow \mathsf{A}^*$ *and* $rm : \mathsf{B} \rightarrow \mathsf{B}^*$*, that permute the values in columns* A *and* B *to create columns* A^* *and* B^**, respectively, such that* $\mathcal{X} : \mathsf{A}^* \sim \mathsf{B}^*$ *is an order compatibility based on Definition 3.0.9 (i.e., in the lexicographic sense).*

We now break down general OCs into several cases depending on the nature of the functions *lm* and *rm*.

Definition 4.1.2 *We say that* $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$ *is a* Syntactic Order Compatibility*,* ***SynOC*** *for short, if* lm *and* rm *are both the identity function. This reduces to the lexicographic notion of order compatibility discussed by Szlichta et al. [9].*

Example 4.1.1 { }: grade_num ∼ grade_letter *is a valid SynOC in Table 1.1.*

The time complexity of validating whether a given *SynOC* holds is linear in the number of tuples in the table [9]. We discuss *SynOC* validation in Section 4.3.

Definition 4.1.3 *We say that* $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$ *is a* Special Case Semantic Order Compatibility*,* $\boldsymbol{SemOC_{sc}}$ *for short, and we show it as* $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}^*$*, if only* lm *is the identity function.*

Table 4.1: A table containing a 3-fan-out, corresponding to Figure 4.3

A	B
a	1
a	2
a	3
b	2
c	1
d	3

Example 4.1.2 { }: grade_num ∼ grade_desc* *is a valid SemOC$_{sc}$ in Table 1.1. If the table is sorted by the syntactic (numeric) order of* grade_num*, there exists a permutation of the values of* grade_desc *that gives a semantic order: [Fail, Good, Very Good, Excellent]. On the other hand,* { }: grade_num ∼ grade_desc* *is not valid in Table 4.2. However, fixing* country *as the context attribute and then inspecting the order relationship between* grade_num *and* grade_desc *yields a valid SemOC$_{sc}$ in the form of* {country}: grade_num ∼ grade_desc*.

We present an algorithm for validating a given *SemOC$_{sc}$* candidate in Section 4.4. The time complexity of this algorithm is polynomial in the number of tuples in the table. This will be shown in Section 4.6.

Definition 4.1.4 *We say that* $\mathcal{X}$: A ∼ B *is a* General Case Semantic Order Compatibility*,* ***SemOC$_{gc}$*** *for short, and we show it as* $\mathcal{X}$: A* ∼ B*, if neither* lm *nor* rm *is the identity function.*

Example 4.1.3 *The SemOC$_{gc}$* { }: g_month* ∼ p_month* *holds in Table 1.2. As we showed in the Introduction, there is a joint (non-lexicographic) ordering of both attributes, and this ordering satisfies Definition 4.1.4.*

Example 4.1.4 *There is no SemOC$_{gc}$ between* A *and* B *in Table 4.1. In other words, there are no functions rm and lm that could permute the values to satisfy Definition 4.1.4.*

Theorem 2 *The problem of validating a given SemOC$_{gc}$ candidate is NP-complete.*

proof sketch: As we will see in Section 4.5, in order to validate a given *SemOC$_{gc}$* candidate over a given relation, we may have to solve what we define below as the *Chain Polarity Problem (CPP)*. We will show that CPP is NP-complete. This entails that the problem of validating a given *SemOC$_{gc}$* candidate is also NP-complete.

Definition 4.1.5 ***The Chain Polarity Problem****: For the* Chain Polarity Problem (CPP)*, the* structure *is a collection of lists of elements. Each list is constrained such that no element appears twice in the list. A list can be interpreted as defining a total order over its elements; e.g. list [a, b, c, d] infers* $a < b$, $b < c$, *and* $c < d$.

A polarization *of the list collection is a new list collection in which, for each list in the original, the list appears or the reverse of the list does.*

The decision question (the problem) for CPP is whether there exists a polarization of the CPP instance such that the union of the total orders represented by the polarizations lists is a strong partial order.

Lemma 1 *(CPP is NP-complete) The Chain Polarization Problem is NP-Complete.*

Proof
The input size of a CPP instance may be measured as the sum of the lengths of its lists; let this be n. Consider a pair explicitly implied by the list collection to be in the binary ordering relation if the pair of elements appears immediately adjacent in one of the lists. Thus, the number of explicitly implied pairs is bounded by n.

CPP is in the class NP. An answer of yes to the corresponding decision question means there exists a polarization of the CPP instance that admits a strong partial order. Given such a polarization witness, its validity can be checked in polynomial time. The size of the polarization is at most n. The set of explicitly implied ordered-relation pairs is at most n. Computing the transitive closure over this set of pairs is then polynomial in n. If no reflexive pair e.g., aa is discovered, then there are no cycles in the transitive closure, and thus this represents a strong partial order. Otherwise, not.

CPP is NP-complete. The known NP-complete problem NAE-3SAT (Not-All-Equal 3SAT) can be reduced to CPP.

The structure of a NAE-3SAT instance is a collection of clauses. Each clause consists of three literals. A literal is a propositional variable or the negation thereof. A clause is interpreted as the disjunction of its literals, and the overall instance is interpreted as the logical formula which is the conjunction of its clauses. Since each clause is of fixed size, the size of the NAE-3SAT instance may be measured by its number of clauses; call this n.

The decision question for NAE-3SAT is whether there exists a truth assignment to the propositional variables (propositions) that satisfies the instance formula such that, for each clause, at least one of its literals is false in the truth assignment and at least one is true (as necessary, of course, for the instance formula to be satisfied). (This extra condition on each of the clauses is the not-all-equal restriction.)

We can establish a mapping from NAE-3SAT instances to *CPP instances which is polynomial time to compute and for which the decision questions are synonymous.

Let $p_1, .., p_m$ be the propositions of the NAE-3SAT instance. For clause i, let $(L_{i,1}, L_{i,2}, L_{i,3})$ represent it, where each $L_{i,j}$ is a placeholder representing the corresponding proposition or negated proposition as according to the clause.

We build a corresponding CPP instance as follows. For each clause, we add three lists. For clause i, add

$[X_{i,1}, a_i, b_i, Y_{i,1}]$, $[X_{i,2}, b_i, c_i, Y_{i,2}]$, and $[X_{i,3}, c_i, a_i, Y_{i,3}]$. Each $X_{i,j}$ and $Y_{i,j}$ are placeholders above, and correspond to the $L_{i,j}$s in the clauses.

We replace them in the lists as follows. There are two cases for each: $L_{i,j}$ corresponds to a proposition or the negation thereof. Without loss of generality, let $L_{i,j}$ correspond to pk or to $\neg pk$. If $L_{i,j}$ corresponds to pk, we replace $X_{i,j}$ with element tk and $Y_{i,j}$ with element fk. Else ($L_{i,j}$ corresponds to $\neg pk$), we replace $X_{i,j}$ with fk and $Y_{i,j}$ with tk.

Call the t_i and f_i elements propositional elements, and call the a_i, b_i, and c_i elements confounder elements.

There exists a polarization of the CPP instance that admits a strong partial order iff the corresponding NAE-3SAT instance is satisfiable such that, for each clause, at least one of its literals has been assigned false.

For the propositional elements, let us interpret $t_i < f_i$ in the partial order as assigning proposition p as true; and $f_i < t_i$ as assigning it false.

For any polarization of the CPP instance that admits a strong partial order, for each clause, i, one or two, but not all three, of the corresponding lists must have been reversed. Otherwise, there will be a cycle in the ordering relation over the confounder elements, a_i, b_i, and c_i.

Since not all three lists for clause i can be reversed, $X_{i,1} < Y_{i,1}, X_{i,2} < Y_{i,2}$, or $X_{i,3} < Y_{i,3}$. And thus, at least one of the clauses literals is marked as true, and so the clause is true. Since at least one of the three lists for clause i must be reversed, we know that $Y_{i,1} < X_{i,1}, Y_{i,2} < X_{i,2}$, or $Y_{i,3} < X_{i,3}$. And thus, not all of the clauses literals are marked as true.

If two lists (coming from different clauses) contradict that is, one implies, say, $t_i < f_i$ and the other that $f_i < t_i$ then there would be a cycle in the transitive closure of the ordering relation. This would contradict that our polarization admits a strong partial order. Thus, for each p_i, the partial order either has $t_i f_i$ or $f_i < t_i$. This corresponds to a truth assignment that satisfies the NAE-3SAT instance.

If no polarization of the CPP instance admits a strong partial order, then there is no truth assignment that satisfies the NAE-3SAT instance such that, for each clause, at least one of its literals has been assigned false. □

Note that every $SynOC$ is valid $SemOC_{sc}$ and every $SemOCsc$ is a valid $SemOC_{gc}$.

Validating a semantic OC candidate amounts to finding a permutation of the values of the left-hand-side and right-hand-side attributes in the OC (i.e., the functions lm and rm) that yields a valid partial order. For a $SynOC$, this permutation and the functions lm and rm are given by the syntactic order over the values of the attributes. For OCs in Definitions 4.1.3 and 4.1.4, we propose new validation algorithms. An algorithm for validating a given $SemOC_{gc}$ candidate is given in Section 4.5. We do not avoid the NP-complete cases in our algorithm. Instead, we set a limit on the time spent validating a $SemOC_{gc}$ candidate and we skip the validation of that particular candidate if the limit is exceeded. However, based on the validity of the other candidates, we can in some cases infer the validity of the original candidate. This will be further explained in the upcoming sections.

4.2 Discovering Semantic Order Compatibilities: Lattice Traversal

In order to find all valid order compatibilities in a given dataset, we traverse the set containment lattice of attributes to generate OC candidates. A lattice corresponding to Table 4.2 is shown in Figure 4.1. The main body of our OC discovery algorithm is shown in Algorithm 1, in which we traverse the lattice level by level, generate candidates (Line 7), validate candidates (Line 5), and prune future candidates depending on the validity of the current candidate (Line 6)[1].

As shown in Algorithm 2, within the function *computeOCs*, each candidate is first tested for validity as a $SynOC$ (Line 3), if invalid it is tested for $SemOCsc$ (Line 5), and if still invalid it is tested for validity as a $SemOCgc$ (Line 7).

Example 4.2.1 *Consider Table 4.2 and the corresponding set-containment lattice shown in Figure 4.1. Below, we show some examples of OCs and FDs that were found at various nodes in the lattice.*

- ***Cell number 9**:*

[1]We use the same pruning rules as the OD discovery algorithm by Szlichta et al. [9], which are based on the axiomatization of the set-based canonical form of ODs.

Table 4.2: Grades in different scales in Canada and in Iran

#	country	grade_num	grade_letter	grade_desc
1	Iran	20.00	A	Excellent
2	Iran	19.00	A	Excellent
3	Iran	12.00	C	Good
4	Iran	10.00	D	Pass
5	Iran	9.00	F	Fail
6	Canada	95.00	A	Excellent
7	Canada	80.00	B	Very Good
8	Canada	75.00	B	Very Good
9	Canada	70.00	C	Good
10	Canada	20.00	F	Fail
11	Canada	12.00	F	Fail

  - $SemOC_{sc}$: $\{\ \}$: $\mathsf{GL} \sim \mathsf{GD}^*$,
  - *FD:* $\{\mathsf{GD}\}$: $[\,] \mapsto_{cst} \mathsf{GL}$
  - *FD:* $\{\mathsf{GL}\}$: $[\,] \mapsto_{cst} \mathsf{GD}$

- ***Cell number 13:***
  - $SemOC_{sc}$: $\{\mathsf{C}\}$: $\mathsf{GN} \sim \mathsf{GD}^*$
  - *FD:* $\{\mathsf{C}, \mathsf{GN}\}$: $[\,] \mapsto_{cst} \mathsf{GD}$

- ***Cell number 14:***
  - *SynOC:* $\{\mathsf{C}\}$: $\mathsf{GN} \sim \mathsf{GL}$

4.3 Validating Syntactic Order Compatiblities

A syntactic order compatibility, or a *SynOC*, refers to an OC studied in previous work, i.e., one that relies on syntactic order over both the left-hand-side and the right-hand-side attributes. We use the algorithm from Szlichta et al. [9] to validate *SynOC*s when traversing the search space.

According to [9], an OC of the form $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$ does not hold over $\mathbf{r}$ if there is a *swap* with respect to the OC.

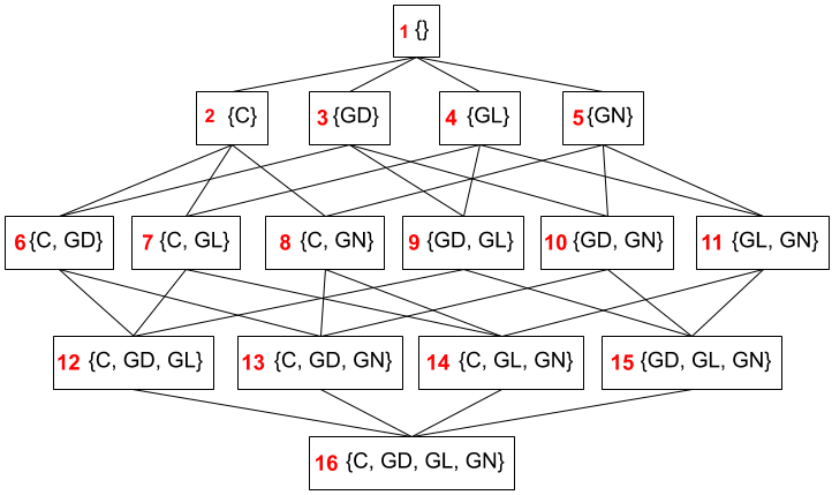


Figure 4.1: Set containment lattice for the attributes in Table 4.2 ($\mathbf{R} = \{\mathsf{country}, \mathsf{grade\_desc}, \mathsf{grade\_letter}, \mathsf{grade\_num}\}$)

Definition 4.3.1 *([9], Definition 5) A* swap *with respect to* $\mathbf{X} \sim \mathbf{Y}$ *is a pair of tuples* s *and* t *such that* $s \prec_{\mathbf{X}} t$, *but* $t \prec_{\mathbf{Y}} s$.

Example 4.3.1 *In Table 1.1 there is a swap between tuples* t1 *and* t6 *as* $t1_{\{\mathsf{grade\_num}\}} < t6_{\{\mathsf{grade\_num}\}}$, *but* $t1_{\{\mathsf{grade\_desc}\}} < t6_{\{\mathsf{grade\_desc}\}}$. *Therefore the* SynOC $\mathcal{X}$: grade_num $\sim$ grade_desc *does not hold. For the first 5 tuples in the table, any time a tuple has a smaller value of* grade_num *compared to another tuple, it also has a smaller value on* grade_desc *(w.r.t. the lexicographic order over the attribute) compared to that tuple. However* t6 *breaks this pattern.*

For *SynOC* validation (Line 3 in Algorithm 2), we use the algorithm in [9], which performs a single scan over the sorted partitions of the attributes involved in a *SynOC* and looks for swaps. If there are no swaps, the OC is valid.

4.4 Validating Semantic Order Compatibilities: Special Case

In order to validate a $SemOC_{sc}$ candidate in the form of $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}^*$ we take the following steps.

Algorithm 1 discoverOCs

Input: Relation $\mathbf{r}$ over schema $\mathbf{R}$
Output: Set of OCs $\mathcal{M}$, such that $\mathbf{r} \models \mathcal{M}$

$\mathcal{L}_0 = \{\}$;
$l = 1$; $\mathcal{L}_1 = \{\mathsf{A} \mid \mathsf{A} \in \mathbf{R}\}$; $\forall_{\mathsf{A} \in R}$
prunedList($\{\}$) $= \{\}$
while $\mathcal{L}_l \neq \{\}$ **do**
    computeOCs($\mathcal{L}_l$)
    pruneLevels($\mathcal{L}_l$)
    $\mathcal{L}_{l+1}$ = *calculateNextLevel*($\mathcal{L}_l$)
    $l = l + 1$
end while
return $\mathcal{M}$

Algorithm 2 computeOCs

for all $\mathcal{X} \in \mathcal{L}_l$ **do**
    for each pair of attributes $\{\mathsf{A}, \mathsf{B}\} \notin$ prunedList($\mathcal{X}$) **do**
        if $\mathcal{X}\backslash\{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$ **then**
            break
        else if $\mathcal{X}\backslash\{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}^*$ or $\mathcal{X}\backslash\{\mathsf{A}, \mathsf{B}\} : \mathsf{B} \sim \mathsf{A}^*$ **then**
            break
        else if $\mathcal{X}\backslash\{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A}^* \sim \mathsf{B}^*$ **then**
            break
        end if
    end for
end for

4.4.1 Validating Each Equivalence Class

Given a $SemOC_{sc}$ candidate $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}^*$, for each equivalence class in $\mathcal{X}$, we order the equivalence classes in $\Pi_{\{\mathsf{B}\}}$ based on their earliest occurrence with sorted equivalence classes of A. We refer to this new sorted partition on B as τ_{B^*}. With this new order on the equivalence classes of B, we look for *swaps*. A swap w.r.t a $SemOC$ candidate $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}^*$ occurs when for tuples t and s : $t_{\{\mathsf{B}^*\}} < s_{\{\mathsf{B}^*\}}$, but $t_{\{\mathsf{A}\}} < s_{\{\mathsf{A}\}}$.

Example 4.4.1 *Consider Table 4.3 and the $SemOC_{sc}$ candidate* $\{\}$: g_month $\sim$ p_month*. *The order of equivalence classes of* p_month *w.r.t* g_month *is [Dey, Bahman, Esfand, ..., Azar]. We refer to this new ordering over* p_month *as* $\tau_{\mathsf{p\_month}^*}$. *Although "Dey" occurs with "12" as well, it has been placed in the order based on its first occurence when sorted*

Table 4.3: numeric representation of Gregorian months and their overlap with Persian months

row	g_month	p_month
1	1	Dey
2	1	Bahman
3	2	Bahman
4	2	Esfand
...	...	...
23	12	Azar
24	12	Dey

Table 4.4: Table 4.3 transformed

row	g_month	p_month*
1	1	0
2	1	1
3	2	1
4	2	2
...	...	...
23	12	11
24	12	0

by g_month. *In other words, the syntactic order on* g_month *determines the function rm :* p_month $\rightarrow$ p_month*. *This function maps values [Dey, Bahman, Esfand, ..., Azar] to values [0, 1, 2, ..., 11]. Table 4.4 shows the transformed version of Table 4.3 after the function rm is applied to the values of* p_month. *Notice that there is a swap between tuples* t1 *and* t24 *as* $\text{t1}_{\{\text{p\_month*}\}} < \text{t24}_{\{\text{p\_month*}\}}$, *but* $\text{t1}_{\{\text{g\_month}\}} < \text{t24}_{\{\text{g\_month}\}}$.

Example 4.4.2 *Consider example 5.2.1 and Table 5.4 in Section 5.2. Table 5.4, as transformed by the function rm :* age_range $\rightarrow$ age_range*, *is shown in Table 4.5. There are no swaps in the transformed table with respect to* {}: age $\sim$ age_range*, *therefore it is a valid order compatibility.*

Table 4.5: Table 5.4 transformed

age	age_range*
15	0
[18, ..., 25]	1
26	1
[26, ..., 40]	2
41	2
[41, ..., 65]	3
66	3
[66, ..., 105]	4

4.4.2 Validating $SemOC_{sc}$ Over All Equivalence Classes

The partial order on B derived from each equivalence class in $\Pi_{\mathcal{X}}$ will be the order of equivalence classes in the sorted partition $\tau_{\mathcal{X}\mathsf{B}^*}$. Unlike individual partial orders for $SemOC_{gc}$, $SemOC_{sc}$ partial orders have a fixed polarization as they have been determined by the syntactic order on A and this syntactic order will be the same in all equivalence classes in $\Pi_{\mathcal{X}}$. Therefore, to obtain the overall order over attribute B in $\mathcal{X}$: A $\sim$ B*, we take the union of all the partial orders as they were derived from each equivalence class without having to fix the polarization of each one.

After all such partial orders are merged, we check the validity of the resulting order. To do so, we check for cycles in the graph representation of the order. The order is valid if the graph is cycle-free.

Example 4.4.3 *Consider Table 4.2 and the $SemOC_{sc}$ candidate* {country}: grade_num $\sim$ grade_desc*. *The function rm* : grade_desc $\rightarrow$ grade_desc* *maps values [Fail, Pass, Good, Very Good, Excellent] to values [0, 1, 2, 3, 4]. Since there are no swaps w.r.t.* {country}: grade_num $\sim$ grade_desc*, *it is a valid OC, and the order over column* grade_desc *as imposed by column* grade_num *is the one specified by function rm.*

4.5 Validating Semantic Order Compatibilities: General Case

In order to validate an OC candidate in the form of $\mathcal{X}$: A $\sim$ B as a $SemOC_{gc}$, we take the following steps, and we state theorems along the way on conditions for invalidity of the

candidate.

4.5.1 Validating a single equivalence class

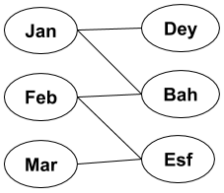
For each equivalence class $\varepsilon(t_x)$, we create a bipartite graph, referred to as *biGraph*, representing how the values of A and B appear together in the table. In the biGraph, one part represents the values of A (equivalence classes in $\Pi_{\mathcal{X}\mathsf{A}}$), and the other represents the values of B (equivalence classes in $\Pi_{\mathcal{X}\mathsf{B}}$). If t_A and t_B appear together in tuple t, then there is an edge in the biGraph between the nodes representing $\varepsilon(t_A)$ and $\varepsilon(t_B)$.

Example 4.5.1 *Figure 4.2 shows the set of biGraphs created for each* $\varepsilon(t_x)$ *in Table 4.6 for candidate* {g_year}: g_month $\sim$ p_month.

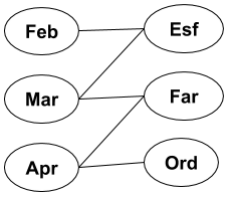
Table 4.6: Gregorian and Persian months corresponding to a set of dates

g_date	g_year	g_month	p_month
2017-01-15	2017	January	Dey
2017-01-25	2017	January	Bahman
2017-02-10	2017	February	Bahman
2017-02-26	2017	February	Esfand
2017-03-01	2017	March	Esfand
2018-02-27	2018	February	Esfand
2018-03-05	2018	March	Esfand
2018-03-28	2018	March	Farvardin
2018-04-18	2018	April	Farvardin
2018-04-25	2018	April	Ordibehesht
2019-04-17	2019	April	Farvardin
2019-04-28	2019	April	Ordibehesht
2019-05-01	2019	May	Ordibehesht
2019-05-22	2019	May	Khordad
2019-11-26	2019	November	Azar
2019-12-05	2019	December	Azar
2019-01-28	2019	December	Dey

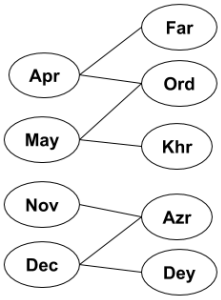
If a biGraph has a node that is connected to at least 3 nodes, each with degree 2, we say that the biGraph has a *3-fan-out*. Figure 4.3 shows the biGraph corresponding to Table 4.1 and it shows the simplest form of a 3-fan-out.



(a) Year 2017



(b) Year 2018



(c) Year 2019

Figure 4.2: biGraphs representing co-occurence of values of *gmonth* and *pmonth* from Table 4.6 for each equivalence class in $\{gyear\}$

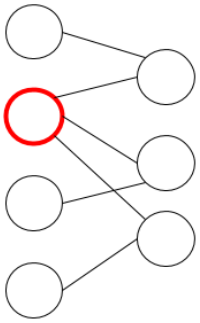


Figure 4.3: biGraph corresponding to Table 4.1

Theorem 3 *Given a single equivalence class $\varepsilon(t_x)$ for the OC candidate $\mathcal{X}$: A $\sim$ B, and the biGraph created to represent the co-occurrence of values of A and B, there exists a valid $SemOC_{gc}$ between A and B in the given equivalence class, if the two following conditions hold:*

1. *The biGraph does not contain a 3-fan-out.*
2. *The biGraph is acyclic.*

proof sketch: We show that if the biGraph is acyclic or if it includes a 3-fan-out, then it is impossible to establish a valid order over the two attributes. A cyclic biGraph represents a table where for each of the attributes the derived partial order, i.e. the corresponding side of the biGraph for the particular attribute, will be cyclic and therefore there will be a swap in the relation with respect to the order compatibility candidate.
If the biGraph includes a 3-fan-out, then there will be an element such as a under A in Table 4.1 with 3 other elements, b, c, d that need to appear immediately adjacent to a in the partial order for the candidate to provide a valid partial order over the table. Let us say that b is to appear immediately before a, and c immediately after a. Then d cannot be placed adjacent to a and any other placement would create a swap in the table with respect to the given OC candidate.

Example 4.5.2 *Consider Table 4.1. The biGraph corresponding to this table is shown in Figure 4.3. This biGraph has a 3-fan-out, therefore the OC $\mathcal{X}$: A $\sim$ B* is not valid. In other words, there does not exist a pair of permutation functions, lm and rm, that could induce a valid OC with respect to Definition 4.1.4.*

Nodes of degree 1 cannot invalidate the biGraph w.r.t. Theorem 3. Therefore, in order to validate a biGraph based on Theorem 3, instead of creating the full biGraph as described earlier, we create a *pruned* version of it we will refer to as *biGraph*$'$. In the pruned version, when encountering values of A or B that only occur with one value of the other attribute, we do not include them in biGraph$'$. Instead, we add them to a set of *singleton* nodes. In our implementation, every node object has 3 features: (1) *singletons:* the set of nodes of degree 1 that it is connected to (2) *connections:* nodes of degree 2 or higher connected to this node (3) *side:* side of the attribute associated with this node in the OC candidate (e.g. in $\mathcal{X}$: A $\sim$ B, nodes representing values of A have side = 0 and nodes representing values of B have side = 1). For example, node "1" in Figure 4.3 has the following features: {singletons: {"c"}, side: 1, connections: {"a"}}.

On the resulting biGraph′, we check for cycles using BFS to satisfy the second condition in Theorem 3. Furthermore, to check for the first condition in the theorem, when visiting each node in the BFS, if the degree of the node is equal to or greater than 3, the candidate is deemed as invalid. Checking for degree of 3 or higher in the biGraph′ is equivalent to checking for a 3-fan-out in the original biGraph, as the singleton nodes were removed from biGraph to create biGraph′.

4.5.2 Chain Derivation

An equivalence class that is valid w.r.t. a $SemOC_{gc}$ in the form of $\mathcal{X}$: A $\sim$ B based on Theorem 3, yields a set of pairs of partial orders we refer to as *chains*, representing the order over the values of A and B. For example, for g_year = 2017, ⟨["January", "February", "March"], ["Dey", "Bah", "Esf"]⟩ is a pair of partial orders derived from the biGraph in Figure 4.2a for the OC {g_year}: g_month $\sim$ p_month. The first entry in the pair represents the order on g_month and the second represents the order on p_month. Considering all chains from all equivalence classes, we do not know if the final order the pair will be as mentioned above, or if both partial orders in the pair above would have to be flipped to look like ⟨["March", "February", "January"], ["Esf", "Bah", "Dey"]⟩. In either case, the two chains in each pair corresponding to a single biGraph are tied to each other and in order for them to be valid over the dataset, their polarization would have to remain fixed w.r.t. each other.

Algorithm 3 describes how chains are derived from a set of biGraphs. We traverse a given biGraph and derive a *chain* which contains two lists denoting the partial order derived from the left-hand-side and right-hand-side of the biGraph. Line 4 makes sure that all components in the biGraph get traversed in case the graph is disconnected. The while loop in 5 helps traverse the main line in the biGraph until we reach the end of the chain. there are two entries in *chain* each corresponding to one side of the biGraph. We switch between different sides by considering left-hand-side at index 0 and right-hand-side at index 1 and using XOR to switch indices. Line 6 adds *currNode* to corresponding entry in *chain*. Line 8 add any nodes with degree 1 that *currNode* is connected to to the opposite side of *chain*. Finally Lines 9 and 11 remove the visited nodes from *unvisitedList* and Line 13 adds the derived *chain* to the set of all chains.

Consider the biGraphs in Figure 4.2 from Example 4.5.1 as inputs to Algorithm 3. The chains derived from each biGraph are presented below.

1. **4.2a**: ⟨[January, February , March] , [Dey, Bahman, Farvardin]⟩

Algorithm 3 deriveChains

Input: Relation **r** over schema **R**

Output: Pairs of chains in the form of $\langle chain_{LHS}, chain_{RHS} \rangle$ kept in *allChains*

```
1: set currNode to a node with degree 1
2: set unvisitedList to set of all nodes
3: chain = ⟨[], []⟩
4: while !unvisitedList.isEmpty() do
5:    while currNode.degree >= 2 do
6:       append node to chain.getSide(currNode.side)
7:       if currNode.singletons.isEmpty() == false then
8:          append the set currNode.singletons to
                        chain.getSide(currNode.side XOR 1)
9:          remove all currNode.singletons from unvisitedList
10:      end if
11:      remove currNode from unvisitedList
12:   end while
13:   add chain to allChains
14: end while
```

2. **4.2b**: ⟨[February, March , April] , [Esfand, Farvardin, Ordibehesht]⟩
3. **4.2c**: ⟨[April, May] , [Farvardin, Ordibehesht, Khordad]⟩
4. **4.2c**: ⟨[November, December] , [Azar, Dey]⟩

An entry such as ⟨[January, February , March] , [Dey, Bahman, Farvardin]⟩, means that [January, February , March] is the chain (partial order) derived from the left-hand-side of the biGraph and [Dey, Bahman, Farvardin] is the one derived from the right-hand-side of the biGraph. The two chains are coupled together, which means if one is reversed but not the other, they no longer represent the biGraph they were derived from.

In order to enforce the polarization of some chains to be fixed w.r.t. each other, we define the notion of polarity groups.

Definition 4.5.1 *For an OC candidate* $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$*, we define a* polarity group $\mathcal{PG}$ *as a group of partial orders bound together. This entails that if a member of this group is reversed, all members of the group have to be reversed so the polarities remain aligned.*

Definition 4.5.2 *The reverse of a polarity group* $\mathcal{PG}$ *is created by reversing all the edges in the LHS and the RHS partial order graphs in the polarity group, and we refer to it as* $\mathcal{PG}'$*.*

All partial orders in a polarity group have a fixed polarization w.r.t all other members of the group. Flipping the polarization of one member, and not the rest, will cause the partial orders to be invalid over the dataset. That is not necessarily the case when looking across different polarity groups.

Starting with pairs of chains derived in Algorithm 3, our aim is to discover as total an order as possible over the values of A and B in $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$. In order to do so, we need to merge individual pairs of chains to create bigger partial orders while checking for validity of the order over the dataset at each step. We make use of the notion of polarity groups and create a set of polarity groups, each including partial orders from one pair of chains. Therefore, initially we start with as many polarity groups as the number of chains derived from the dataset, and we proceed to *merge* polarity groups to create bigger orders step by step. This procedure will be described in detail in Section 4.5.3.

4.5.3 Merge and Validate Polarity Groups

We start with a set of polarity groups, each initialized with a pair of chains derived from a single biGraph$'$. Algorithm 4 takes this set of polarity groups as input, and iterating over pairs of polarity groups, it merges them only if the merge produces a valid order in one polarization or the other, but not both (Lines 9 to 14). The first step to checking if two poalrity groups can be merged in a fixed polarization is to check whether they have more than one element in common on either right-hand-side or left-hand-side of the polarity group (Lines 4 to 6). The algorithm continues to loop over pairs of polarity groups until no more merging can be done, or until the candidate is deemed invalid.

Specifically, we merge $\mathcal{PG}_i$ and $\mathcal{PG}_j$ to create *mergeStraight* and then merge $\mathcal{PG}_i$ and $\mathcal{PG}_j^{'}$ to create *mergeReverse*, and we test both for cycles. *mergeStraight* is the graph corresponding to the partial order yielded by merging two polarity groups in their given polarization. *mergeReverse* reverses one of the two polarity groups and then merges them. If only one of *mergeStraight* or *mergeReverse* is cycle-free, we take that as the final merge result between $\mathcal{PG}_i$ and $\mathcal{PG}_j$. Otherwise, if both are cycle-free, we do not proceed with the merge as it cannot be performed in a fixed polarization (Line 17). If both merges are cyclic, the algorithm returns "Invalid" for the candidate at hand (Line 15). We check for cyclicity of the graphs corresponding to the partial order in each polarity group as a proxy for the validity of the partial order.

The graph *looseConns* keeps information about the polarity groups who have two or more elements in common but they cannot be merged in a fixed polarization, or those who have only one element in common. This information is kept in the form of polarity group

indices i and j. Algorithm 5 will traverse this graph to check whether there can be a valid order established over all the polarity groups with *loose* connections. If the data presents an NP-complete problem, it will be dealt with in Algorithm 5 which will be discussed in more detail in the next section.

Algorithm 4 mergePolarityGroups

Input: A set of polarity groups $\{\mathcal{PG}_1, ..., \mathcal{PG}_m\}$
Output: A set of polarity groups

empty graph *looseConns*, *nextToCheck* = {}, *currToCheck* = $\{1, 2, ..., m\}$
while *!currToCheck.isEmpty()* **do**
  for each pair of polarity groups $\mathcal{PG}_i$ and $\mathcal{PG}_j$ with i and j being in *currToCheck* **do**
    $common_{LHS} = |\mathcal{PG}_i[0] \cap \mathcal{PG}_j[0]|$
    $common_{RHS} = |\mathcal{PG}_i[1] \cap \mathcal{PG}_j[1]|$
    if $common_{LHS} >= 2$ or $common_{RHS} >= 2$ **then**
      mergeStraight = *merge*($\mathcal{PG}_i$, $\mathcal{PG}_j$)
      mergeReverse = *merge*($\mathcal{PG}_i$, $\mathcal{PG}'_j$)
      if *mergeStraight.isCyclic()* and *!mergeReverse.isCyclic()* **then**
        $\mathcal{PG}_i$ = *mergeReverse*, add i to *nextToCheck*
        $\mathcal{PG}_j$ = {}, remove edge (i, j) from *looseConns*
      else if *!mergeStraight.isCyclic()* and *mergeReverse.isCyclic()* **then**
        $\mathcal{PG}_i$ = *mergeStraight*, add i to *nextToCheck*
        $\mathcal{PG}_j$ = {}, remove edge (i, j) from *looseConns*
      else if *mergeStraight.isCyclic()* and *mergeReverse.isCyclic()* **then**
        return Invalid
      else
        add edge (i, j) to *looseConns*
      end if
    else if ($common_{LHS} == 1$) or ($common_{RHS} == 1$) **then**
      add edge (i, j) to *looseConns*
    end if
  end for
  currToCheck = *nextToCheck*, *nextToCheck* = {}
end while
return set of polarity groups $\{\mathcal{PG}_1, ..., \mathcal{PG}_m\}$

4.5.4 Merge Polarity Groups With Loose Connections

Given the *looseConns* graph from Algorithm 4, Algorithm 5 determines if there is a permutation of the polarizations of polarity groups in this graph such that the resulting partial

order from all polarity groups will be valid over the dataset.

Line 2 checks if *looseConns* is cyclic. If it is cycle-free then we can just return the set of polarity groups, as any permutation of their polarization would yield a valid partial order over the dataset. If *looseConns* is cyclic then the algorithm proceeds with a BFS traversal of this graph (Line 6). Every node in *looseConns* represents the index of a polarity group. Every node (i.e. or every polarity group) that is visited is pushed into *mainStack* (Line 8). Every entry in the stack represents a polarity group and the polarization with which it should be merged with other polarity groups in the stack. For example if -4 is an entry in the stack, it means that polarity group with index 4 needs to be merged with the other polarity groups in the stack in the reverse of its polarization ($\mathcal{PG}_4^{'}$). Then all the polarity groups corresponding to the contents of the graph are merged and tested for validity (Lines 10 to 13). If the contents of the stack do not yield a valid partial order (the graph representing the partial order corresponding to the merge is cyclic), the contents of the stack are popped to arrange a different unchecked permutation of polarizations for polarity groups and then the cycle is repeated until either all permutations are checked and deemed to be invalid, or all nodes in the *looseConns* graph have been successfully visited.

4.6 Complexity Analysis

In this section, we explain the worst-case complexity of our algorithms. As described in Section 4.2, we consider the different types of OCs in the following order:

1. $SynOC$
2. $SemOC_{sc}$
3. $SemOC_{gc}$

We show that the validation becomes more complex from $SynOC$ validation to $SemOC_{gc}$ validation. Validation of a single $SynOC$ candidate is linear in the number of tuples, as discussed in FASTOD [9].

4.6.1 Validation of a Single $SemOC_{sc}$

Our approach to validating a single $SemOC_{sc}$ candidate involves ordering the values of the right-hand-side attribute by the syntactic order of the left-hand-side attribute, which

costs $tlog(t)$, with t being the number of tuples. Checking for swaps involves a single scan over the newly assigned order, which costs $O(t)$. Merging of all chains and checking the representative graph of the resulting partial order also costs $O(t)$. Therefore this algorithm is in $O(tlog(t))$.

4.6.2 Validation of a Single $SemOC_{gc}$

Validation of a single $SemOC_{gc}$ candidate involves three major parts: *deriveChains*, *mergePolarityGroups*, and *handleLooseConns* shown in Algorithms 3 to 5.

First, we scan the tuples to create the biGraph$'$. Then, we use BFS to validate the biGraph$'$. Algorithm 3 involves a single scan over each biGraph$'$ to derive the chains for the left-hand-side and right-hand-side of the OC candidate. Therefore, up to this point, the complexity is linear in the number of tuples.

Moving on to Algorithm 4, the *while* loop in Line 2 is repeated until no more merging can be done. In the worst case, the number of polarity groups that need to be checked in Line 3 within each iteration of the outer while loop is on the order of n, with n being the number of polarity groups at the start of 4, making the number of pairs to be checked $O(n^2)$. This means that the while loop is repeated $O(\log n)$ times. Every calculation of the intersection between $\mathcal{PG}_i$ and $\mathcal{PG}_j$ in Lines 4 and 5 costs $O(m_i + m_j)$, with m_i and m_j being the number of edges in the corresponding partial orders for $\mathcal{PG}_i$ and $\mathcal{PG}_j$, respectively. Furthermore,for every merging of two polarity groups in Lines 7 and 8, the cost is $O(max(m_i, m_j))$. Consider the complexity of Lines 4 to 8 to be in $O(t)$, with t being the total number of tuples. n itself can be in $O(t)$, therefore Algorithm 4 is in $O(t^3log(t))$.

The complexity of Algorithm 5 depends on the number of polarity groups that are connected in the *looseConns* graph that is passed on to it, and the structure of the graph. In the worst case, the algorithm has to check for the validity of all possible merges of polarity groups with different polarizations. The cost of doing so would be exponential in the number of polarity groups in the *looseConns* graph. As we will observe in Section 5.3, the exponential complexity of this algorithm causes the validation of certain candidates to take up to hours when experimenting with a real world dataset. In order to ensure that the algorithm moves on from such candidates, we set a limit of 400 on the number of different combinations that can be checked in Algorithm 5. If this limit is exceeded, the algorithm avoids making a decision about the validity of the candidate. Therefore, in practice, our algorithm may not be complete in the discovery of all minimal OCs. However, there might be ways to compensate for this when searching the next level of the lattice for valid candidates, as explained below.

Example 4.6.1 *Figure 4.4 shows parts of a lattice corresponding to the schema* $\mathbf{R} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$. *Assume that the lattice traversal is inspecting cell number 19 in the lattice, and when the algorithm attempts to validate the* $SemOC_{gc}$ *candidate* $\{\mathsf{A}\}$: $\mathsf{B}^* \sim \mathsf{E}^*$, *the limit on the number of iterations is exceeded and the validation is temrinated. The algorithm cannot yet decide on the validity of this OC candidate. However, while inspecting cells 28 and 29, respectively, OC candidates* $\{\mathsf{A}, \mathsf{C}\}$: $\mathsf{B} \sim \mathsf{E}$ *and* $\{\mathsf{A}, \mathsf{D}\}$: $\mathsf{B} \sim \mathsf{E}$ *are checked for validity as described in Algorithm 2. If any of these two candidates are found to be invalid on this level of the lattice, then we know that the original candidate inspected in cell 19 must also have been invalid.*

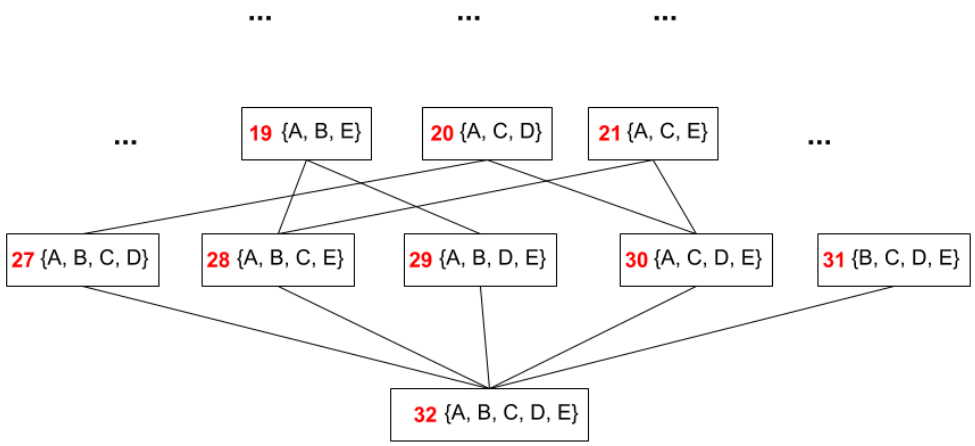


Figure 4.4: Set containment lattice for $\mathbf{R} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$

In Section 5.3.2, we will show how many OC candidates cannot be validated on real datasets due to an exceedingly high number of polarity groups.

Algorithm 5 handleLooseConns

Input: *looseConns* graph, set of polarity groups $\{\mathcal{PG}_1, ..., \mathcal{PG}_m\}$ after Algorithm 4 is finished
Output: A graph representing an order over the two attributes

```
1: Stack mainStack = [], Stack sideStack = []
2: if !looseConns.isCyclic() then
3:    // any polarization of the existing polarity groups would result in a valid order over the
      dataset
4:    return {PG_1, ..., PG_m}
5: else
6:    for every node i in looseConns visited in a BFS manner do
7:       // i is an index to a polarity group in the set of polarity groups
8:       mainStack.push(i)
9:       mergeRes = {}
10:      for every k in mainStack do
11:         mergeRes = merge(mergeRes, PG_k)
12:      end for
13:      while mergeRes.isCyclic() do
14:         popped = mainStack.pop()
15:         while True do
16:            if mainStack.isEmpty() then
17:               return "Invalid"
18:            else if popped < 0 then
19:               sideStack.push(−1 ∗ popped)
20:               popped = mainStack.pop()
21:            else if popped > 0 then
22:               mainStack.push(−1 ∗ popped)
23:               while !sideStack.isEmpty() do
24:                  mainStack.push(sideStack.pop())
25:               end while
26:               break
27:            end if
28:         end while
29:      end while
30:   end for
31: end if
```

Chapter 5

Experiments

We now present qualitative experiments, focusing on interesting orders found in real-life datasets, and quantitative experiments, focusing on the performance and scalability of discovering semantic ODs in real-life datasets.

We implemented the discovery algorithm on top of a Java code base developed by the authors of FASTOD [9]. We rely on the code base for the validation of $SynOC$s. Validation methods for other OCs have been developed as part of our work.

The experiments were done on a 64-bit Linux machine with Ubuntu 16.04, 16 GiB RAM, and an intel core i7-6770HQ CPU with a frequency of 2.60 GHz.

Table 5.1: $SemOC_{gc}$ candidates from the *flights* dataset exceeding 400 iterations in their validation

row	$SemOC_{gc}$ **candidate**	**# polarity groups**
1	{DayofMonth, FlightNum}: Carrier ∼ OriginAirportID	26
2	{DayofMonth, FlightNum}: FlightDate ∼ OriginAirportID	51
3	{DayofMonth, FlightNum}: UniqueCarrier ∼ OriginAirportID	26
4	{FlightDate, FlightNum}: TailNum ∼ OriginAirportID	48
5	{DayofMonth, FlightNum}: AirlineID ∼ OriginAirportID	26
6	{DayofMonth, FlightNum, OriginAirportID}: DayofMonth ∼ TailNum	35

Table 5.2: skipped candidates from rows 1 and 2 of Table 5.1, as considered on the next lattice level

row in Table 5.1	**candidate**	**type**	**valid?**
1	-{DayofMonth, FlightNum, Month}: Carrier ∼ OriginAirportID	$SemOC_{gc}$	F
2	-{DayofMonth, TailNum, FlightNum}: FlightDate ∼ OriginAirportID	$SynOC$	T
	-{DayofMonth, AirlineID, FlightNum}: OriginAirportID ∼ FlightDate	$SemOC_{sc}$	T
	-{DayofMonth, UniqueCarrier, FlightNum}: OriginAirportID ∼ FlightDate	$SemOC_{sc}$	T
	-{DayofMonth, Carrier, FlightNum}: OriginAirportID ∼ FlightDate	$SemOC_{sc}$	T
3	-{Month, DayofMonth, FlightNum}: UniqueCarrier ∼ OriginAirportID	$SemOC_{gc}$	F
	-{DayofMonth, DayOfWeek, FlightNum}: UniqueCarrier ∼ OriginAirportID	$SemOC_{gc}$	F
4	-{FlightDate, UniqueCarrier, FlightNum}: TailNum ∼ OriginAirportID	$SynOC$	F
	-{FlightDate, AirlineID, FlightNum}: TailNum ∼ OriginAirportID	$SynOC$	F
	-{FlightDate, Carrier, FlightNum}: TailNum ∼ OriginAirportID	$SemOC_{sc}$	F
5	-{Month, DayofMonth, FlightNum}: AilrineID ∼ OriginAirportID	$SemOC_{gc}$	F
	-{DayofMonth, DayofWeek, FlightNum}: AilrineID ∼ OriginAirportID	$SemOC_{gc}$	F
6	-{Month, AirlineID, FlightNum, OriginAirportID}: DayofMonth ∼ TailNum	$SynOC$	F

5.1 Data Characteristics

The datasets used for the experiments are available through the Hasso Plattner Institute (HPI) repository[1] and the UCI Machine Learning repository[2]. These datasets are:

- **ncvoter**: This dataset has 22 columns and around 940K rows, and contains voter registration information from the state of North Carolina. Each row includes information about an individual registered to vote in the state of North Carolina such as district, name, address, phone number, age, registration date, race, political affiliation, and sex.
- **flights**: This dataset has 20 columns and 500K rows, and contains information on U.S. domestic flights. Each row includes information about a single flight such as flight date, carrier, origin airport ID, origin city, origin state, and origin WAC.
- **fd-reduced-30**: This synthetic dataset has 30 columns and 250K rows. It was generated by the dbtesma[3] data generator.
- **adult**: This dataset has 14 columns and around 49K rows. It contains census data collected in 1994. Each row includes information about an individual such as their age, education, employment status, marital status, occupation, race, capital gain, capital loss, native country, and hours per week of work.
- **olympic athletes**: This dataset has 9 columns and around 3K rows. It contains data on olympic athletes. Each row includes information about an Olympic athlete such as name, height in cm, height in ft, weight, country code, sex, and the event (e.g. Men's Volleyball).

5.2 Qualitative Experiments

Example 5.2.1 *Consider Table 5.4 that shows the age range assigned to each age from a dataset of registered voters in the state of North Carolina. There does not hold an FD on this dataset, as not all voters with ages 26, 41, and 66 have been assigned to the same age range as the others in their age, and also each age range encompasses a number of different ages. There also does not hold an OC, based on the definition of OCs in FASTOD[9],*

[1] http://metanome.de
[2] https://archive.ics.uci.edu
[3] https://sourceforge.net/projects/dbtesma

as sorting the values by age *does not provide an ordering on* age_range *that matches the lexicographic ordering of the column.*

Now consider the $SemOC_{sc}$ candidate {}: age $\sim$ age_range*. The order imposed on column* age_range *by column* age *leads to the following semantic order: ["Age Under 18 Or Invalid Birth Dates", "Age 18-25", "Age 26-40", "Age 41-65", "Age Over 66"]. Thus, the concept of semantic order dependencies allows us to discover an order over a column whose order is not lexicographic, but rather a semantic one such as ranges of age from younger to older.*

Table 5.4: a sample set of ages and their corresponding age range

age	age_range
15	Age Under 18 Or Invalid Birth Dates
[18, ..., 25]	Age 18 - 25
26	Age 18 - 25
[26, ..., 40]	Age 26 - 40
41	Age 26 - 40
[41, ..., 65]	Age 41 - 65
66	Age 41 - 65
[66, ..., 105]	Age Over 66

Example 5.2.2 *Table 5.3 contains a set of dates from the Gregorian calendar and their corresponding month and year in the Persian and the Chinese calendars. Our algorithm discovers* {c_year, p_year}: c_month $\sim$ p_month *as a valid $SemOC_{gc}$. The order implied in the context of* {c_year, p_year} *is as follows:*

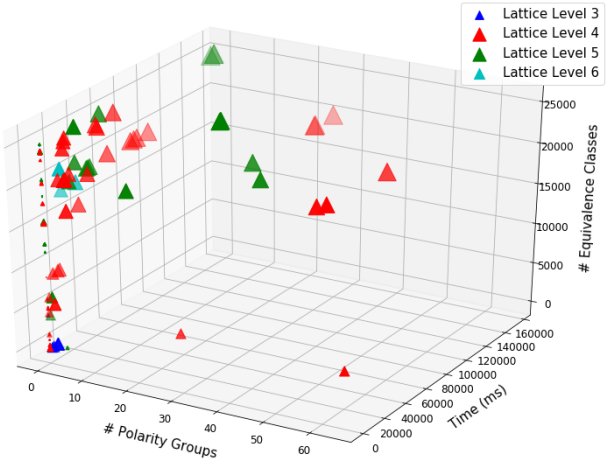
- **c_month***: [Ox, Rat, Pig, Dog, Rooster, Monkey, Sheep, Horse, Snake, Dragon, Hare, Tiger]*
- **p_month***: [Bahman, Dey, Azar, Aban, Mehr, Shahrivar, Mordad, Tir, Khordad, Ordibehesht, Farvardin]*

5.3 Quantitative Experiments

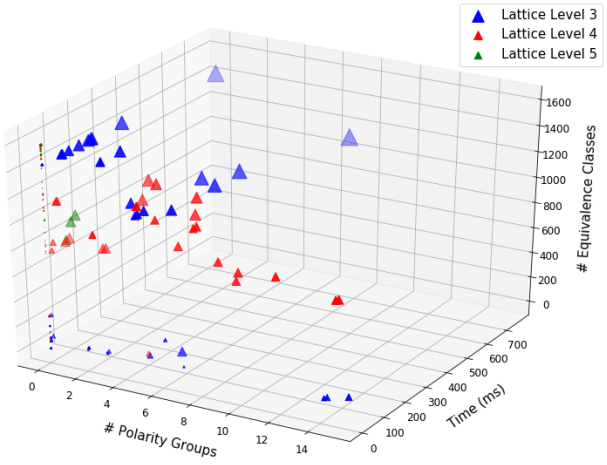
The performance experiments presented in this section are run on each of the 7 datasets described in Section 5.1, on 100k tuples (or on the whole dataset if the total number of tuples is less than 100k), and 10 columns.

5.3.1 $SemOC_{gc}$ Validation

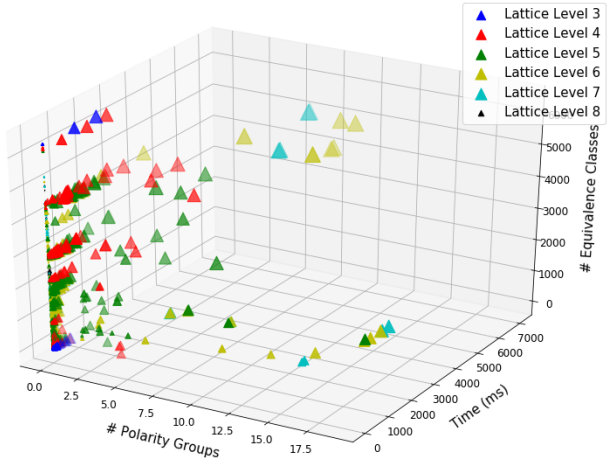
Figure 5.1 shows the performance of $SemOC_{gc}$ validation with respect to the number of polarity groups and the number of equivalence classes of the context attributes. Each of the four plots, from Figure 5.1a to Figure 5.1d, shows a scatter plot for one dataset. One point on this figure represents the time it takes to validate a particular candidate. As the number of equivalence classes or the number of polarity groups increases, it takes a longer to validate a $SemOC_{gc}$ candidate. This is an expected result, as in the first module in the $SemOC_{gc}$ validation algorithm, the computational focus is on the merging of different polarity groups. This module is sensitive to the number of equivalence classes as that determines the initial number of polarity groups, hence the sensitivity of the performance to number of equivalence classes. In the second module in the $SemOC_{gc}$ validation algorithm, we use a brute force technique to determine if there exists a way for all the polarity groups to be oriented and merged in a way that makes a valid partial order. The complexity of module 2 depends on the number of polarity groups.



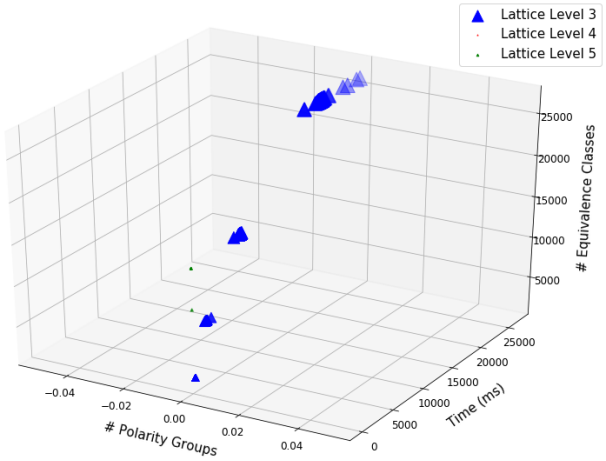
(a) *flights* dataset



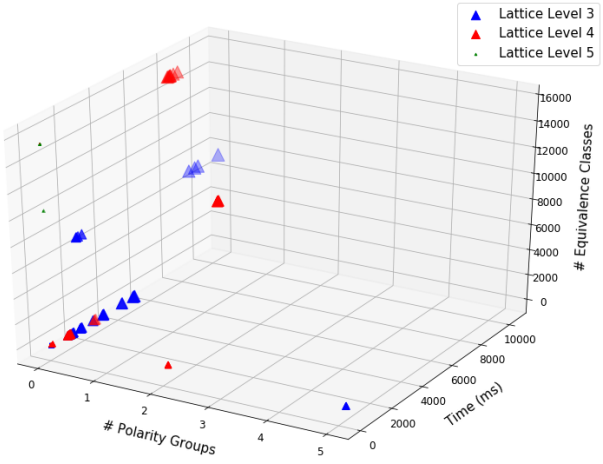
(b) *athlete heights* dataset



(c) *adult* dataset



(d) *fd-reduced-30* dataset



(e) *ncvoter* dataset

Figure 5.1: Scatter plots showing the time taken for each $SemOC_{gc}$ candidate validation.

5.3.2 NP-complete Cases in Practice

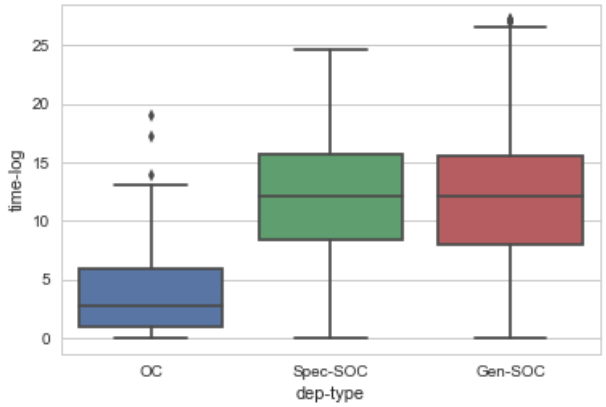
As mentioned in Theorem 2, validating a $SemOC_{gc}$ is NP-complete. The complex cases are characterized by an unusually long validation time spent in the second module of the $SemOC_{gc}$ validation algorithm, i.e. Algorithm 5. Based on our observations, in some cases it takes more than 5 hours for a single $SemOC_{gc}$ validation, after which the program has to be manually terminated. We now check to see how often these cases are encountered in real world datasets.

As discussed earlier, we set a limit of 400 on the number of times the polarization of the polarity groups can be altered, after which our discovery algorithm moves on to the next candidate. The limit of 400 was reached only on the *flights* dataset during the validation of the $SemOC_{gc}$ candidates shown in Table 5.1, along with the number of polarity groups that had to be dealt with by module 2 for each candidate. Out of the 987 total OC candidates analyzed, only the 6 shown in the table exceeded the limit on the number of iterations. As shown in the table, all of the 6 candidates are from level 4 or higher in the lattice.

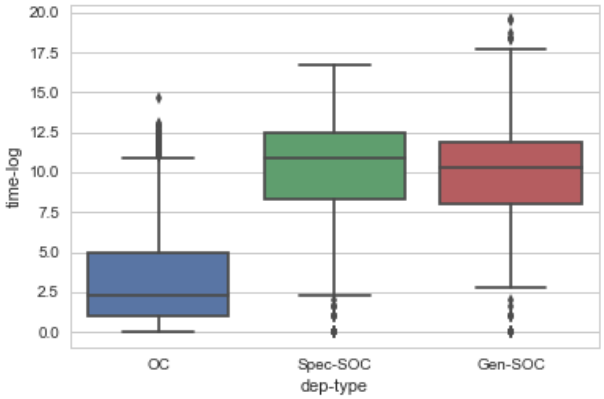
The candidates whose validation exceeded the allowed limit are "re-checked" in the next lattice level, but with a larger context. Table 5.2 shows these larger candidates for each row in Table 5.1, along with their type and if they were found to be valid or not. If at least one of these candidates is invalid, we can infer that the original candidate is invalid. This is the case with for candidates corresponding to rows 1, 3, 4, 5, and 6 of Table 5.1. This allows us to mitigate the potential incompleteness of our approach.

5.3.3 Validation Times for Different Types of Dependencies

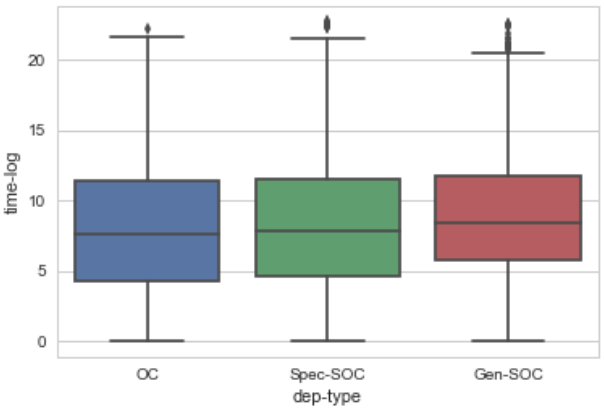
Figure 5.2 depicts a set of boxplots showing the time (logarithmic) it takes for OD candidates of different types to be validated. The dependency types from left to right are $SynOC$, $SemOC_{sc}$, and $SemOC_{gc}$. On average, the time to validate $SemOC_{gc}$s has the biggest variation. As described in Section 4.5, in $SemOC_{gc}$ validation, the overall partial order graph is built by merging smaller components, and after each merge, the validity of the resulting graph is checked. In $SemOC_{sc}$ validation, the overall partial graph is created first by merging all the components, and then it is validated. Therefore, there are more opportunities for the $SemOC_{gc}$ algorithm to terminate if the OC is invalid as it is built gradually and checked for validity at every step of the way, hence the high variation in validation times is an expected observation.



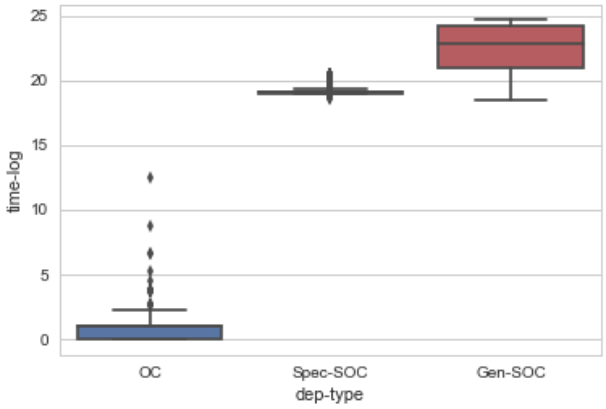
(a) *flights* dataset



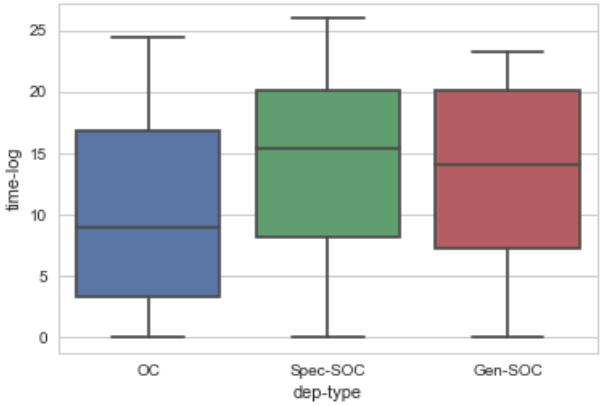
(b) *athlete heights* dataset



(c) *adult* dataset



(d) *fd-reduced-30* dataset



(e) *ncvoter* dataset

Figure 5.2: Boxplots showing the time taken (logarithmic) to validate different types of order candidates on different datasets.

Table 5.3: Overlapping_months from the Persian and Chinese calendar

#	c_month	c_year	p_year	p_month	Gregorian Date
1	Ox	4716	1397	Bahman	2019-01-31
2	Ox	4716	1397	Dey	2019-01-07
3	Rat	4716	1397	Dey	2019-12-31
4	Rat	4716	1397	Azar	2018-12-08
5	Pig	4716	1397	Azar	2018-11-21
6	Pig	4716	1397	Aban	2018-11-08
7	Dog	4716	1397	Aban	2018-10-31
8	Dog	4716	1397	Mehr	2018-10-10
9	Rooster	4716	1397	Mehr	2018-09-30
10	Rooster	4716	1397	Shahrivar	2018-09-10
11	Monkey	4716	1397	Sharivar	2018-08-31
12	Monkey	4716	1397	Mordad	2018-08-12
13	Sheep	4716	1397	Mordad	2018-07-31
14	Sheep	4716	1397	Tir	2018-07-22
15	Horse	4716	1397	Tir	2018-06-30
16	Horse	4716	1397	Khordad	2018-06-21
17	Snake	4716	1397	Khordad	2018-05-31
18	Snake	4716	1397	Ordibehesht	2018-05-21
19	Dragon	4716	1397	Ordibehesht	2018-04-30
20	Dragon	4716	1397	Farvardin	2018-04-17
21	Hare	4716	1397	Farvardin	2018-03-31
22	Hare	4716	1396	Esfand	2018-03-18
23	Tiger	4716	1396	Esfand	2018-02-28
24	Tiger	4716	1396	Bahman	2018-02-17
25	Ox	4715	1396	Bahman	2018-01-31
26	Ox	4715	1396	Dey	2018-01-20
27	Rat	4715	1396	Dey	2017-12-31
28	Rat	4715	1396	Azar	2017-12-19
29	Pig	4715	1396	Azar	2017-11-30
30	Pig	4715	1396	Aban	2017-11-21
31	Dog	4715	1396	Aban	2017-10-31
32	Dog	4715	1396	Mehr	2017-10-22

Chapter 6

Conclusions and Future Work

In this paper, we proposed the notion of semantic order compatibility, abbreviated *SemOC*, which generalizes traditional order compatibility by allowing non-obvious (i.e., non numeric or lexicographic) orders. We showed that validating a *SemOC* is NP-complete in the general case, but such hard cases do not occur often in practice. We also presented and experimentally evaluated an algorithm to discover *SemOC*s from ther data.

A natural direction for future work is to investigate *approximate SemOC*s that hold with some exceptions. This is a non-trival task as we not only have to check the validity of a candidate, but also derive the semantic order. Adding approximation may change the resulting order.

Conditional FDs on databases have been studied by [2] for data cleaning. Defining conditional OCs and *SemOC*s is an open problem.

Our discovery algorithm uses the same pruning rules as the FASTOD algorithm for discovering canonical-form ODs [9]. An interesting direction for future work is to develop additional *SemOC*-specific pruning rules.

References

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. Data profiling. *Synthesis Lectures on Data Management*, 10(4):1–154, 2018.

[2] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd international conference on data engineering*, pages 746–755. IEEE, 2007.

[3] S. Ginsburg and R. Hull. Order Dependency in the Relational Model. *TCS, 26(1): 149-195*, 1983.

[4] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[5] P. Langer and F. Naumann. Efficient order dependency detection. *VLDB J.*, 25(2):223–241, 2016.

[6] Wilfred Ng. An extension of the relational data model to incorporate ordered domains. *ACM Trans. Database Syst.*, 26(3):344–383, September 2001.

[7] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[8] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of Order Dependencies. *PVLDB, 5(11): 1220-1231*, 2012.

[9] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *Proceedings of the VLDB Endowment*, 10(7):721–732, 2017.

[10] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Expressiveness and complexity of order dependencies. *Proceedings of the VLDB Endowment*, 6(14):1858–1869, 2013.